Binary Search Tree

Generated by Doxygen 1.8.11

Contents

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 bstNode Struct Reference

This is a structure for Binary Search Trees represented by a linked list with three fields, one for information storage, one to point to the left child and one for the right child of the tree.

`#include <bst_c.c>`

Collaboration diagram for bstNode:

Data Fields

- int **data**
- struct **bstNode** * **left**
- struct **bstNode** * **right**

3.1.1 Detailed Description

This is a structure for Binary Search Trees represented by a linked list with three fields, one for information storage, one to point to the left child and one for the right child of the tree.

3.1.2 Field Documentation

3.1.2.1 int data

information of one node

3.1.2.2 struct bstNode* left

left child of the node

3.1.2.3 struct bstNode* right

right child of the node

The documentation for this struct was generated from the following file:

- **bst_c.c**

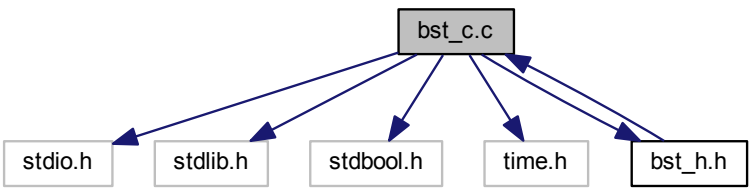
Chapter 4

File Documentation

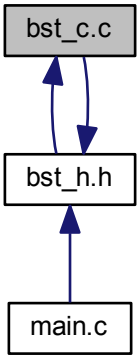
4.1 bst_c.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include "bst_h.h"
```

Include dependency graph for bst_c.c:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct **bstNode**

  This is a structure for Binary Search Trees represented by a linked list with three fields, one for information storage, one to point to the left child and one for the right child of the tree.

Functions

- int **randomize** ()
- struct **bstNode** ∗ **getNode** (int key)

  *A struct **bstNode** (p. 5) procedure for creating a new node.*
- struct **bstNode** ∗ **insert_bst** (struct **bstNode** ∗root, int key)

  *A struct **bstNode** (p. 5) function to insert an element.*
- bool **search_node** (struct **bstNode** ∗root, int key)

  A boolean function to search for an element in the tree.
- struct **bstNode** ∗ **create_bst** (struct **bstNode** ∗root, int nrNode)

  function that generates a binary tree inserting random values in the nodes
- int **bstSrd** (struct **bstNode** ∗root)

  a function that prints the in-order traversal for a tree
- int **bstRsd** (struct **bstNode** ∗root)

  a function that prints the pre-order traversal for a tree
- int **bstSdr** (struct **bstNode** ∗root)

  a function that prints the post-order traversal for a tree
- int **height** (struct **bstNode** ∗root)

  Compute the "height" of a tree the number of nodes along the longest path from the root node down to the farthest leaf node.
- int **printGivenLevel** (struct **bstNode** ∗root, int level)

  a void function that print the nodes from a given level
- void **printLevelOrder** (struct **bstNode** ∗root)

  prints the nodes of the tree in level order
- struct **bstNode** ∗ **findMin** (struct **bstNode** ∗root)

find the smallest value in the tree

- struct **bstNode** ∗ **deleteNode** (struct **bstNode** ∗root, int key)

  function that delete a node which has a value we want to delete
- void **freeTree** (struct **bstNode** ∗root)

  delete a BST
- void **nodeDelete** (struct **bstNode** ∗root, int n)

  a funcrion that delete a given number of elements from a tree

4.1.1 Function Documentation

4.1.1.1 void bstRsd (struct bstNode ∗ *root*)

a function that prints the pre-order traversal for a tree

Parameters

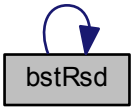
root	a struct **bstNode** (p. 5) variable - address of the root

Returns

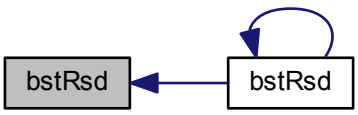
0, when reach the end of the tree

we start by printing the root of every visited node, then visit the left subtree and so with the right subtree

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.1.2 void bstSdr (struct bstNode * root)

a function that prints the post-order traversal for a tree

Parameters

root	a struct **bstNode** (p. 5) variable - address of the root

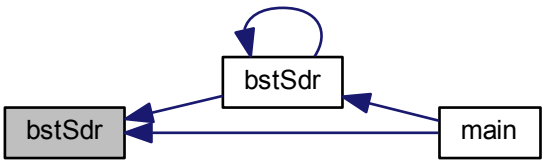
Returns

0, when reach the end of the tree

we to traverse the left subtree of the tree as we arrive at the most left leaf, then go to the right side and print its value

Here is the call graph for this function:

Here is the caller graph for this function:



4.1.1.3 int bstSrd (struct bstNode * root)

a function that prints the in-order traversal for a tree

Parameters

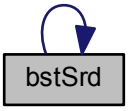
root	a struct **bstNode** (p. 5) variable - address of the first element

Returns

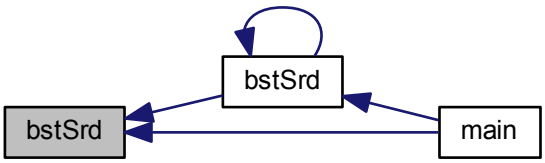
0, when reach the end of the tree

we use recursivity to traverse the left subtree of the tree as we arrive at the most left leaf, we print it, then go to the right side and do the same thing for the right subtree

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.1.4 struct bstNode * create_bst (struct bstNode * *root,* int *nrNode*)

function that generates a binary tree inserting random values in the nodes

Parameters

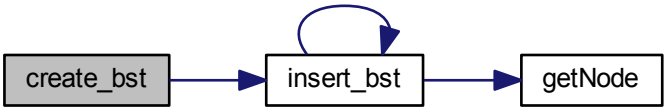
root	of struct **bstNode** (p. 5) type - first element address
nrNode	an integer - the number of nodes in the tree

Returns

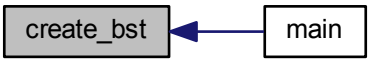
doesn't return anything

we insert nodes in the tree using the generation of random numbers

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.1.5 struct bstNode * deleteNode (struct bstNode * *root*, int *key*)

function that delete a node which has a value we want to delete

Parameters

root	a struct **bstNode** (p. 5) - address of the root
key	an integer - the value of the node we try to delete

Returns

the address of the deleted node

if the tree is empty the function will return the root address

if the value is smaller than the value of the root, then the waned value will be searched in the left subtree

if the value is greater than the value of the root, then the waned value will be searched in the right subtree

we found the wanted value and three cases can be distinguished

1. Case 1 - the node doesn't have any child
2. Case 2 - the node has only one child
3. Case 3 - the node has both left and right children

Case 1: no children

we simply deallocate the node from memory using free function and make it points to NULL

Case 2: one child

there can be 2 situations, the node can have either the left node or a right node, the algorithm is similar for both of them

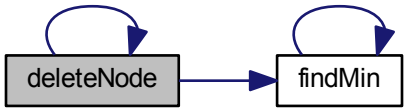
-if the node has a right child, we will use a auxiliary variable which will take the address of the node to be deleted while the node become its right child, the only thing that remains to be done is to deallocate the memory stored in the temporary variable

-if the node has a left child, we will use a auxiliary variable which will take the address of the node to be deleted while the node become its left child, the only thing that remains to be done is to deallocate the memory stored in the temporary variable

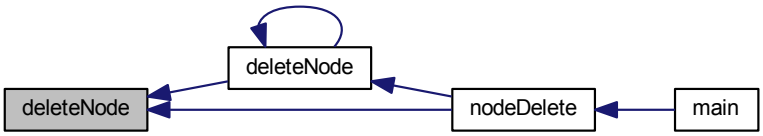
Case 3 - two children

we will use the function findMin to find the minimum value in the right subtree, its address will be stored in the variable temp next step is to copy the value of the minimum value in the node that has the wanted value.Now we have two nodes with the same value and we can be in any case described above so we have to recall the function deleteNode with the parameters root->right, the right child of the current node and root->data

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.1.6 struct bstNode * findMin (struct bstNode * *root*)

find the smallest value in the tree

Parameters

root	a struct **bstNode** (p. 5) variable - address of the first element

Returns

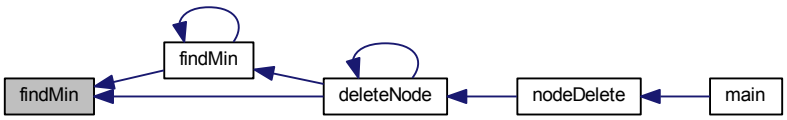
the address of the node with the smallest value

if the tree is empty the function will return 0;

else,we will move in the left subtree while there is a left child

Here is the call graph for this function:

Here is the caller graph for this function:



4.1.1.7 void freeTree (struct bstNode * *root*)

delete a BST

Parameters

root	struct **bstNode** (p. 5) - address of the root

Returns

nothing

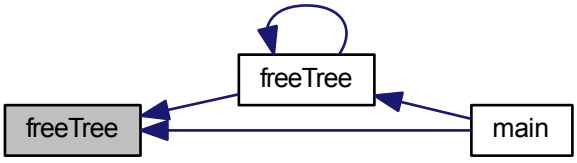
delete the left subtree

delete the right subtree

delete the root

Here is the call graph for this function:

Here is the caller graph for this function:



4.1.1.8 struct bstNode * getNode (int *key*)

A struct **bstNode** (p. 5) procedure for creating a new node.

Parameters

key	an integer - the value of the node

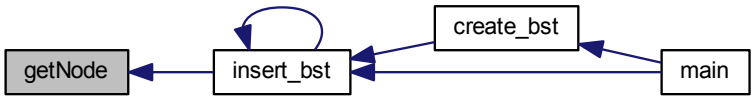
Returns

the address of the new node

the new node takes the value key

because initially the node does not have any children, we make its children to NULL

Here is the caller graph for this function:



4.1.1.9 int height (struct bstNode * *node*)

Compute the "height" of a tree the number of nodes along the longest path from the root node down to the farthest leaf node.

Parameters

root	struct **bstNode** (p. 5) pointer - address of the root

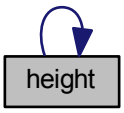
Returns

the height of the tree

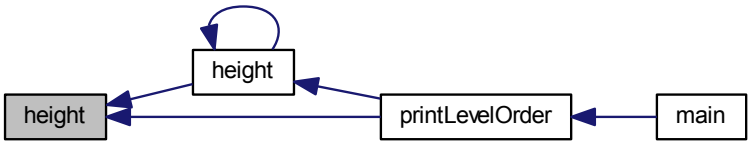
compute the height of each subtree

use the larger one

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.1.10 struct bstNode * insert_bst (struct bstNode * *root*, int *key*)

A struct **bstNode** (p. 5) function to insert an element.

Parameters

root	a pointer of type struct **bstNode** (p. 5) - the address of the root(first element)
key	an integer - the value to be inserted

Returns

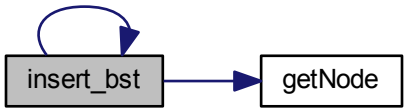
the address of the inserted node

if the node does not exist we create it;

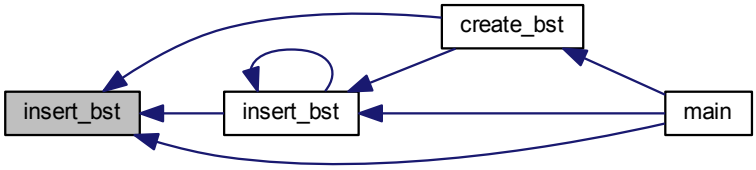
if the value we want to put in the tree is lesser than the root information, we will insert it in the left subtree

otherwise it will be inserted it in the right subtree

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.1.11 int nodeDelete (struct bstNode * *root*, int *n*)

a funcrion that delete a given number of elements from a tree

Parameters

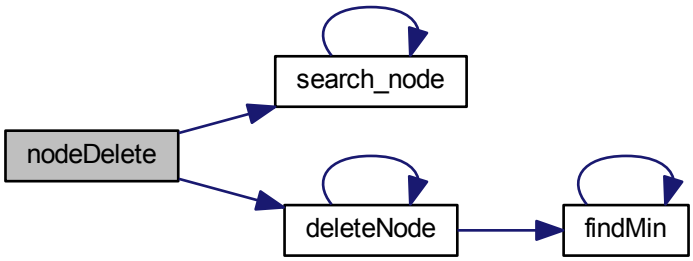
root	struct **bstNode** (p. 5) - the address of the root
n	an integer - the number of nodes to be deleted

Returns

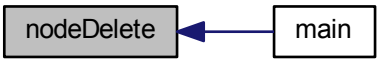
nothing

first we try to find if the node exists in the tree, and we do this by calling the procedure search_node for each value, if data is an element from the tree we delete it, else we return to the point where we generate the information, until the node exists in the tree

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.1.12 void printGivenLevel (struct bstNode * *root*, int *level*)

a void function that print the nodes from a given level

Parameters

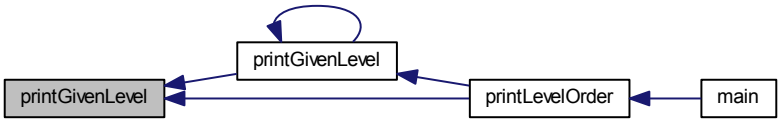
root	struct **bstNode** (p. 5) - address of the root
level	an integer - the level whose nodes we want to print

Returns

nothing

Here is the call graph for this function:

Here is the caller graph for this function:



4.1.1.13 void printLevelOrder (struct **bstNode** * *root*)

prints the nodes of the tree in level order

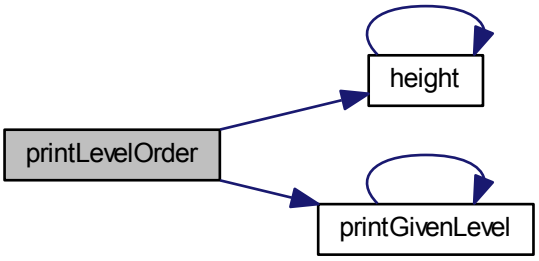
Parameters

root	struct **bstNode** (p. 5) pointer - the address of the first element

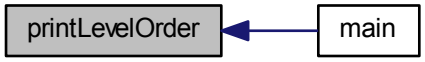
Returns

nothing

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.1.14 int randomize ( )

4.1.1.15 bool search_node (struct **bstNode** * *root*, int *key*)

A boolean function to search for an element in the tree.

Parameters

root	a struct **bstNode** (p. 5) pointer - the address of the root
key	an integer - the value to be searched for

Returns

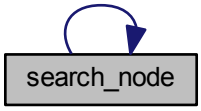
true if the value exist in the tree
false if the value doesn't exist

i found it - the function become true

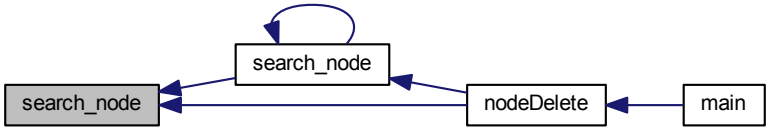
if the value we are looking for is smaller than the root information, we will look for it in the left subtree

otherwise we will look for it in the right subtree

Here is the call graph for this function:

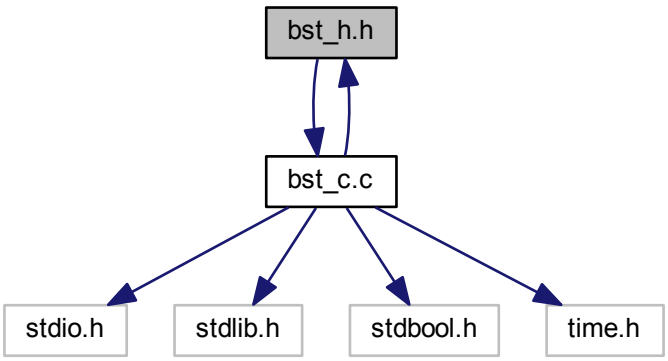


Here is the caller graph for this function:

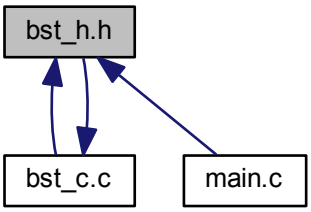


4.2 bst_h.h File Reference

`#include "bst_c.c"`
Include dependency graph for bst_h.h:



This graph shows which files directly or indirectly include this file:



Functions

- struct **bstNode** ∗ **getNode** (int key)

  *A struct **bstNode** (p. 5) procedure for creating a new node.*
- struct **bstNode** ∗ **insert_bst** (struct **bstNode** ∗root, int key)

  *A struct **bstNode** (p. 5) function to insert an element.*
- bool **search_node** (struct **bstNode** ∗root, int key)

  A boolean function to search for an element in the tree.
- struct **bstNode** ∗ **create_bst** (struct **bstNode** ∗root, int nrNode)

  function that generates a binary tree inserting random values in the nodes
- int **bstSrd** (struct **bstNode** ∗root)

  a function that prints the in-order traversal for a tree
- int **bstRsd** (struct **bstNode** ∗root)

  a function that prints the pre-order traversal for a tree
- int **bstSdr** (struct **bstNode** ∗root)

  a function that prints the post-order traversal for a tree
- struct **bstNode** ∗ **findMin** (struct **bstNode** ∗root)

  find the smallest value in the tree
- struct **bstNode** ∗ **deleteNode** (struct **bstNode** ∗root, int key)

  function that delete a node which has a value we want to delete
- int **printGivenLevel** (struct **bstNode** ∗root, int level)

  a void function that print the nodes from a given level
- int **height** (struct **bstNode** ∗root)

  Compute the "height" of a tree the number of nodes along the longest path from the root node down to the farthest leaf node.
- void **printLevelOrder** (struct **bstNode** ∗root)

  prints the nodes of the tree in level order
- void **freeTree** (struct **bstNode** ∗root)

  delete a BST
- void **nodeDelete** (struct **bstNode** ∗root, int n)

  a funcrion that delete a given number of elements from a tree

4.2.1 Function Documentation

4.2.1.1 int bstRsd (struct bstNode ∗ *root*)

a function that prints the pre-order traversal for a tree

Parameters

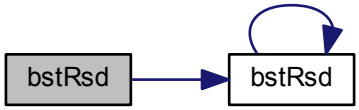
root	a struct **bstNode** (p. 5) variable - address of the root

Returns

0, when reach the end of the tree

we start by printing the root of every visited node, then visit the left subtree and so with the right subtree

Here is the call graph for this function:



Here is the caller graph for this function:

4.2.1.2 int bstSdr (struct bstNode * *root*)

a function that prints the post-order traversal for a tree

Parameters

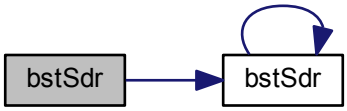
root	a struct **bstNode** (p. 5) variable - address of the root

Returns

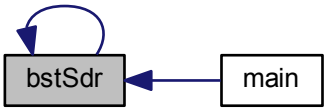
0, when reach the end of the tree

we to traverse the left subtree of the tree as we arrive at the most left leaf, then go to the right side and print its value

Here is the call graph for this function:



Here is the caller graph for this function:



4.2.1.3 int bstSrd (struct **bstNode** * *root*)

a function that prints the in-order traversal for a tree

Parameters

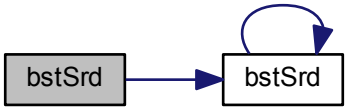
root	a struct **bstNode** (p. 5) variable - address of the first element

Returns

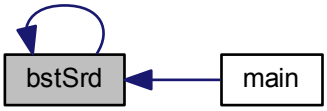
0, when reach the end of the tree

we use recursivity to traverse the left subtree of the tree as we arrive at the most left leaf, we print it, then go to the right side and do the same thing for the right subtree

Here is the call graph for this function:



Here is the caller graph for this function:



4.2.1.4 struct bstNode* create_bst (struct bstNode * *root*, int *nrNode*)

function that generates a binary tree inserting random values in the nodes

Parameters

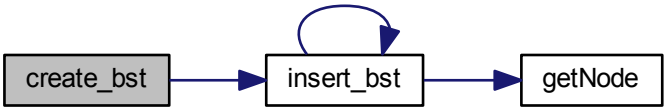
root	of struct **bstNode** (p. 5) type - first element address
nrNode	an integer - the number of nodes in the tree

Returns

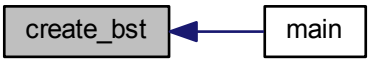
doesn't return anything

we insert nodes in the tree using the generation of random numbers

Here is the call graph for this function:



Here is the caller graph for this function:



4.2.1.5 struct bstNode* deleteNode (struct bstNode * *root,* int *key*)

function that delete a node which has a value we want to delete

Parameters

root	a struct **bstNode** (p. 5) - address of the root
key	an integer - the value of the node we try to delete

Returns

the address of the deleted node

if the tree is empty the function will return the root address

if the value is smaller than the value of the root, then the waned value will be searched in the left subtree

if the value is greater than the value of the root, then the waned value will be searched in the right subtree

we found the wanted value and three cases can be distinguished

1. Case 1 - the node doesn't have any child
2. Case 2 - the node has only one child
3. Case 3 - the node has both left and right children

Case 1: no children

we simply deallocate the node from memory using free function and make it points to NULL

Case 2: one child

there can be 2 situations, the node can have either the left node or a right node, the algorithm is similar for both of them

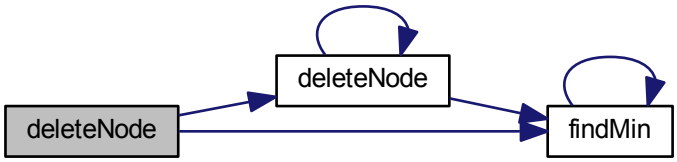
-if the node has a right child, we will use a auxiliary variable which will take the address of the node to be deleted while the node become its right child, the only thing that remains to be done is to deallocate the memory stored in the temporary variable

-if the node has a left child, we will use a auxiliary variable which will take the address of the node to be deleted while the node become its left child, the only thing that remains to be done is to deallocate the memory stored in the temporary variable

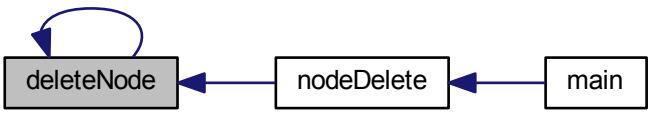
Case 3 - two children

we will use the function findMin to find the minimum value in the right subtree, its address will be stored in the variable temp next step is to copy the value of the minimum value in the node that has the wanted value.Now we have two nodes with the same value and we can be in any case described above so we have to recall the function deleteNode with the parameters root->right, the right child of the current node and root->data

Here is the call graph for this function:



Here is the caller graph for this function:



4.2.1.6 struct bstNode* findMin (struct bstNode * *root*)

find the smallest value in the tree

Parameters

root	a struct **bstNode** (p. 5) variable - address of the first element

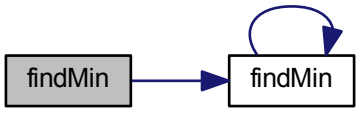
Returns

the address of the node with the smallest value

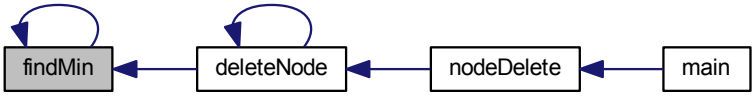
if the tree is empty the function will return 0;

else,we will move in the left subtree while there is a left child

Here is the call graph for this function:



Here is the caller graph for this function:



4.2.1.7 void freeTree (struct bstNode * *root*)

delete a BST

Parameters

root	struct **bstNode** (p. 5) - address of the root

Returns

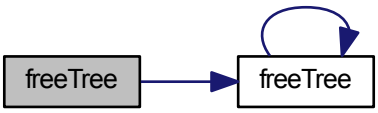
nothing

delete the left subtree

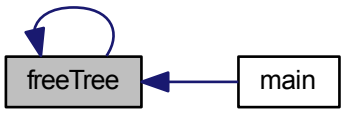
delete the right subtree

delete the root

Here is the call graph for this function:



Here is the caller graph for this function:



4.2.1.8 struct bstNode* getNode (int *key*)

A struct **bstNode** (p. 5) procedure for creating a new node.

Parameters

key	an integer - the value of the node

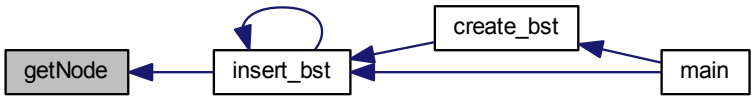
Returns

the address of the new node

the new node takes the value key

because initially the node does not have any children, we make its children to NULL

Here is the caller graph for this function:



4.2.1.9 int height (struct bstNode * *root*)

Compute the "height" of a tree the number of nodes along the longest path from the root node down to the farthest leaf node.

Parameters

root	struct **bstNode** (p. 5) pointer - address of the root

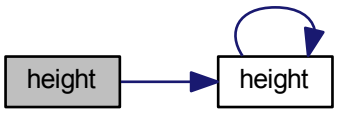
Returns

the height of the tree

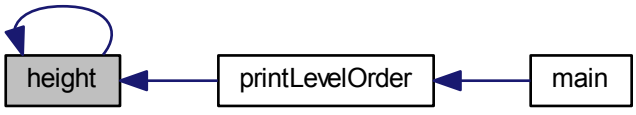
compute the height of each subtree

use the larger one

Here is the call graph for this function:



Here is the caller graph for this function:



4.2.1.10 struct bstNode* insert_bst (struct bstNode * *root,* int *key*)

A struct **bstNode** (p. 5) function to insert an element.

Parameters

root	a pointer of type struct **bstNode** (p. 5) - the address of the root(first element)
key	an integer - the value to be inserted

Returns

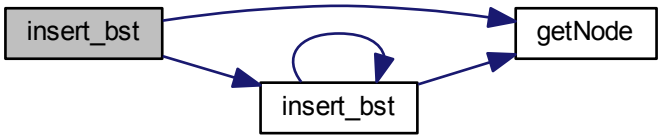
the address of the inserted node

if the node does not exist we create it;

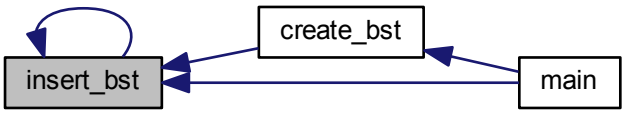
if the value we want to put in the tree is lesser than the root information, we will insert it in the left subtree

otherwise it will be inserted it in the right subtree

Here is the call graph for this function:



Here is the caller graph for this function:



4.2.1.11 void nodeDelete (struct bstNode * *root,* int *n*)

a funcrion that delete a given number of elements from a tree

Parameters

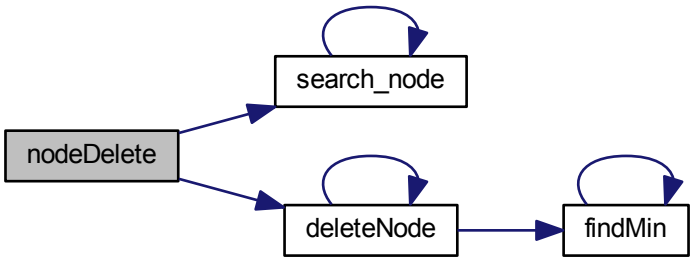
root	struct **bstNode** (p. 5) - the address of the root
n	an integer - the number of nodes to be deleted

Returns

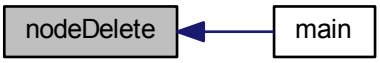
nothing

first we try to find if the node exists in the tree, and we do this by calling the procedure search_node for each value, if data is an element from the tree we delete it, else we return to the point where we generate the information, until the node exists in the tree

Here is the call graph for this function:



Here is the caller graph for this function:



4.2.1.12 int printGivenLevel (struct bstNode * *root*, int *level*)

a void function that print the nodes from a given level

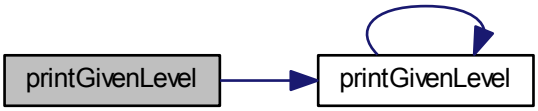
Parameters

root	struct **bstNode** (p. 5) - address of the root
level	an integer - the level whose nodes we want to print

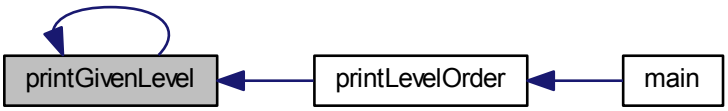
Returns

nothing

Here is the call graph for this function:



Here is the caller graph for this function:



4.2.1.13 void printLevelOrder (struct **bstNode** * *root*)

prints the nodes of the tree in level order

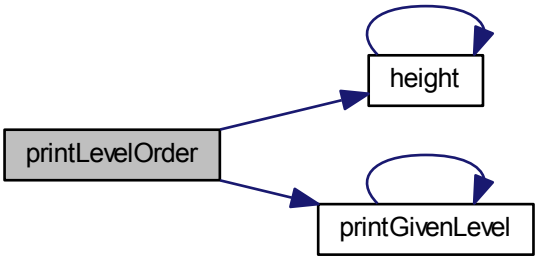
Parameters

root	struct **bstNode** (p. 5) pointer - the address of the first element

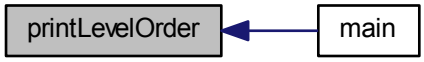
Returns

nothing

Here is the call graph for this function:



Here is the caller graph for this function:



4.2.1.14 bool search_node (struct bstNode * *root*, int *key*)

A boolean function to search for an element in the tree.

Parameters

root	a struct **bstNode** (p. 5) pointer - the address of the root
key	an integer - the value to be searched for

Returns

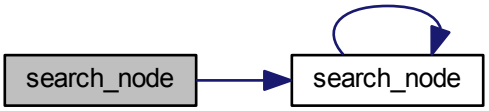
true if the value exist in the tree
false if the value doesn't exist

i found it - the function become true

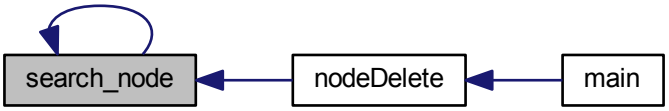
if the value we are looking for is smaller than the root information, we will look for it in the left subtree

otherwise we will look for it in the right subtree

Here is the call graph for this function:

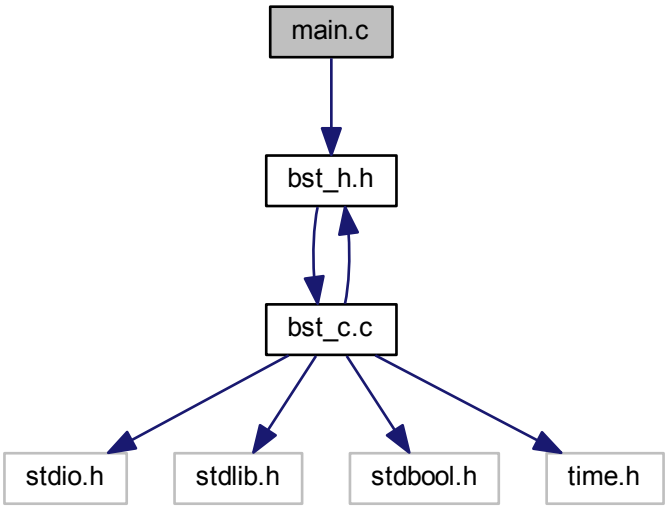


Here is the caller graph for this function:



4.3 main.c File Reference

`#include "bst_h.h"`
Include dependency graph for main.c:



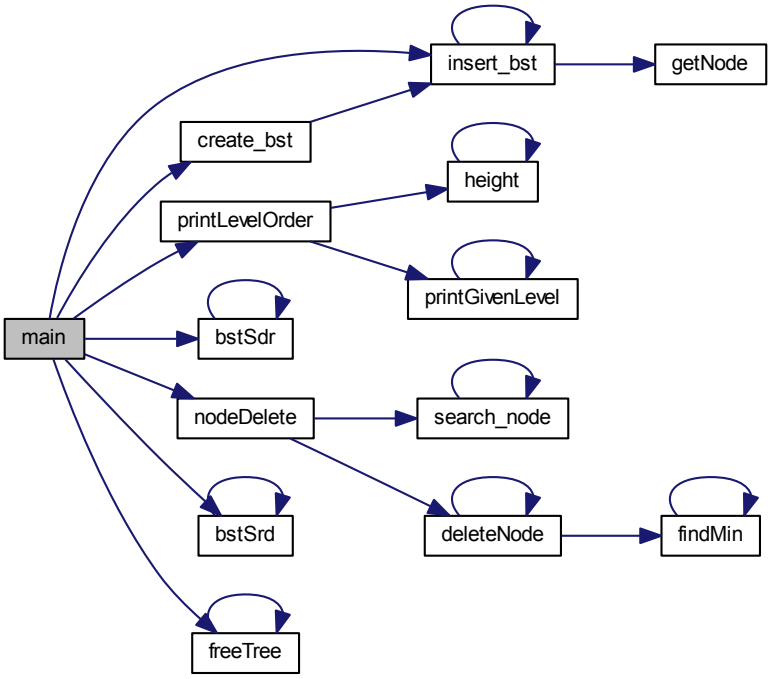
Functions

- int **main** ()

4.3.1 Function Documentation

4.3.1.1 int main ( )

Here is the call graph for this function:



Index